601.445/601.645: Practical Cryptographic Systems	October 11, 2021
Assignment 2	
Instructor: Matthew Green And Alishah Chator	*Due: 11:59 pm October 26, 2021*

Name: ____

The assignment must be completed individually. You are permitted to use the Internet and any printed references, but your code must be your own!

Please submit the completed assignment via Gradescope.

Wireguard[1] is a prominent cryptographic protocol used for establishing encrypted Virtual Private Networks (VPNs). A key building block is the Noise protocol[2], which is used for the handshake. Key features of the Noise protocol are its modular design, emphasis on Diffie-Hellman key agreement, and formal verification. In fact, there are several resources[3] for obtaining machine-generated proofs of security for protocols in the Noise framework. The main Noise message pattern used by many applications is the `IK` pattern:

```
IK:
<- s
...
-> e, es, s, ss
<- e, ee, se
```

This pattern specifies the contents of each handshake message, what operations to perform, and how to update state during the handshake. At the end of the handshake both the initiator and receiver will be able to derive a pair of symmetric keys to use for encrypting communication with each other. The details of what the components of this pattern mean can be found in the Noise specification linked below.

In class we learned of several ways that issues with both the design and implementation of cryptographic primitives and protocols can lead to serious attacks. In this assignment, you will be applying these ideas as you try to compromise a server engaging in a flawed protocol.

Problem 1: Breaking a Bad Protocol Implementation (50 points)

In this problem, you are tasked with compromising the communications between a client and server in order to recover some secret information. It appears that the client and server are using a custom protocol known as wireguard-lite. You were able to find an implementation of this protocol here[4]. As you look over this protocol you notice several key things:

[1] `https://www.wireguard.com/protocol/`
[2] `http://www.noiseprotocol.org/noise.html`
[3] `https://noiseexplorer.com/`
[4] You can find this at `https://github.com/alichator/wg-lite`

1. There seem to be several hardcoded values in the code. However, it appears that these values will be manually changed prior to the client or server running the code. **That is, you can assume there will be hardcoded values but those values may be different on the real client or server compared to the code you found.**

2. You notice the protocol uses a modified version of the Noise `IK` pattern that they refer to as `IKSign`. This pattern replaces the `se` token with a `sign` token which represents a ECDSA signature[5][6] over the message contents. So this pattern looks like the following:
```
IKSign:
<- s
...
-> e, es, s, ss
<- e, ee, sign
```
This handshake protocol involves a preshared static public key from the server for both Diffie Hellman and ECDSA operations. How these keys are generated may be of interest.

3. This code also appears to have many dubious implementation decisions, such as custom random number generators and multiple different elliptic curves (curve25519 and P256) used in different places.

4. The protocol that the client and server are engaging in appears to be quite simple. After the handshake concludes a client can send a `normal_request` or a `secret` query to get some hardcoded response from the server. You notice that the client appears to be hardcoded to only send the `normal_request`.

For this problem you will produce a Go program that interacts with the client and server in order to obtain the server's response to the `secret` query from the client. This will likely require being able to decrypt and modify messages that are being sent between the two parties.

While normally a protocol such as this would happen over a network, for simplicity in this assignment, we will assume that the parties are communicating through files. Each run of the client or the server will require providing several pieces of information to facilitate this. First, which **step** of the protocol they are on. There are 3 possible steps with the first run of the client starting on step 1, and then the server running on step 1. Then the client and server run again on step 2. Finally, the client runs on step 3 and ends the protocol. Additionally, the **seed** that the server uses for its random number generator will be required. Finally, you must provide the files that the client and server will read and write from. Thus the syntax for running the client or server will be:

```
wg-lite <client or server> <server's seed> <protocol step> <filename for outgoing
message> <filename of incoming message from step 1> <filename of incoming message
from step 2>
```

[5] https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm
[6] https://blog.trailofbits.com/2020/06/11/ecdsa-handle-with-care/

So for an example seed value of 1, the correct sequence of program calls would look like:

```
wg-lite client 1 1 client-message-1 server-message-1 server-message-2
wg-lite server 1 1 server-message-1 client-message-1 client-message-2
wg-lite client 1 2 client-message-2 server-message-1 server-message-2
wg-lite server 1 2 server-message-2 client-message-1 client-message-2
wg-lite client 1 3 client-message-3 server-message-1 server-message-2
```

Your code can call this program with any arguments that you would like in the process of carrying out your attack. Additionally, you may repurpose any code from the repositories linked in this document. Additionally, the `math/big.Int` library may be of use. The command line interface of your program should be:

```
compromiser <path to wg-lite>
```

Your program should output a single line (no newline at the end) that is the recovered response to the client's `secret` query.

Note: While you can modify the code provided for testing purposes, keep in mind that the `wg-lite` program used by the autograder will look exactly as the code appears in the repository with the exception of different hardcoded values.

Written Problems

1. Consider a protocol using the specified `IK` pattern for its handshake. This protocol supports "Zero-RTT" where the first handshake message's payload is encrypted and authenticated. What does it mean for a request to be **idempotent** and why might it be a bad idea to include a non-idempotent request in the first payload.

2. IPSec[7] is another tunneling protocol with similar goals to Wireguard. Briefly compare and contrast these protocols

[7] https://en.wikipedia.org/wiki/IPsec